Notes on Costas Array Search and Local Signature Transport

Bogdan Dumitru, Cristian Budala*

Abstract

We report three computational results for Costas array search. First, we describe and benchmark an order-28 search based on prefix decomposition, symmetry reduction, forward checking, and a limited two-row support test. Second, we describe a CUDA split-and-conquer implementation for order 30 in which the GPU expands reduced prefixes into independent work units and assigns each subtree to one warp. The measured throughput projects to 7.97 single-GPU years for the complete order-30 search. Third, we use local difference patterns from smaller Costas arrays to restrict searches at larger orders. These restricted searches quickly returned 5,512 valid Costas arrays for 63 of the tested order pairs. In particular, patterns from orders 26 and 27 produced valid Costas arrays of order 28.

MSC 2020: 05B10, 05A05, 68W10.

1 Introduction

An order-n Costas array is an $n \times n$ board with one dot in each row and each column, such that the vector between any two dots is different from the vector between any other two dots. Costas arrays arose from sonar/radar waveform design and are now studied as combinatorial objects with algebraic constructions; exhaustive counts are known only for relatively small orders [6, 12, 8, 11]. Equivalently, if the array is written as a permutation

$$p = (p_0, p_1, \ldots, p_{n-1}),$$

then for every horizontal displacement d, the differences

$$p_{i+d} - p_i, \qquad 0 \le i < n - d,$$

are nonzero and pairwise distinct.

The enumerations of Costas arrays of orders 28 and 29 required large cluster computations [9, 10]. Their reported CPU times suggested that, many years later, orders 30 or 32 might be approachable with similar methods on modern hardware. In practice, reproducing order-28-scale performance depends on implementation choices that are not fixed by the mathematical backtracking problem alone: prefix partitioning, symmetry handling, forward checking, local pruning, and low-level representation. Earlier search work already made clear that symmetry and lookahead mechanisms are central to high-order Costas enumeration [4, 14].

We focus on implementation and measured performance rather than proposing a new backtracking algorithm. We validate the solvers against known counts and report the benchmark protocol in Appendix A.

The solver source code is available in the accompanying GitHub repo.[1] External Costas array databases are used only as comparison data and are cited explicitly where relevant [3].

*Faculty of Mathematics and Computer Science, University of Bucharest, Romania. Emails: bogdan.dumitru@fmi.unibuc.ro, cristian.budala1@gmail.com

[1] https://github.com/bogdan27182/costas-paper

2 Reconstructing a Competitive Order-28 Search

The CPU solver performs a depth-first search over permutations of $\{0, \ldots, n-1\}$. Columns are filled in the fixed order

$$0, 1, \ldots, n-1.$$

For order 28, we use

$$n = 28, \qquad m = 6, \qquad b = 31,$$

where m is the prefix depth. The value $b = 31$ is an offset used to encode signed vertical differences in 64-bit masks: a difference δ is stored in bit $b + \delta$, so negative differences have nonnegative bit positions. The computation first generates all valid depth-m prefix states satisfying the D_4 first-entry constraints described below. It then shuffles these prefixes with a fixed pseudo-random seed and solves a sample of the resulting independent subtrees.

2.1 Search State and Forward Checking

The search fills columns from left to right. At depth k, the solver has placed

$$p_0, \ldots, p_{k-1}.$$

The Costas condition can be checked one horizontal displacement at a time: for a fixed gap h, the differences $p_i - p_{i-h}$ must be pairwise distinct. These values form the h-th row of the difference triangle [2]. The implementation maintains three state components.

First, it stores the partial permutation itself. Second, it stores, for each horizontal displacement h, a set of already used vertical differences:

$$D_h = \{p_i - p_{i-h} : h \le i < k\}, \qquad 1 \le h < k.$$

For $n = 28$, each D_h is a 64-bit mask. A signed difference δ is stored in bit $b + \delta$, with $b = 31$. Thus the bit for the difference $u - v$ is

$$2^{b+u-v}.$$

The value mask for a row value x is 2^x.

Third, for each future column $c \ge k$, the solver stores a forbidden-value mask F_c. Bit x is set in F_c if x is already used, if placing x at column c would immediately repeat a Costas vector with one of the already placed dots, or if x is excluded by an active first-entry symmetry constraint. Let S_c denote the set of values excluded by those symmetry constraints, with $S_c = \emptyset$ when none is active.

The invariant is:

$$x \notin F_c \iff x \text{ is not ruled out at column } c \text{ by a used row, an existing difference, or an active symmetry constraint.}$$

Equivalently, after $p_0, \ldots, p_{k-1}$ have been placed,

$$F_c = S_c \cup \{p_0, \ldots, p_{k-1}\} \cup \{x : x - p_i \in D_{c-i} \text{ for some } 0 \le i < k\}, \qquad c \ge k.$$

When a value v is placed at column k, the solver first checks and inserts the new differences

$$v - p_{k-d}, \qquad 1 \le d \le k,$$

into D_d. If any of these differences is already present, the candidate is rejected. Otherwise, for every future column $c > k$, the new mask is

$$F_c \leftarrow F_c \cup \{v\} \cup \{x : x - v \in D_{c-k}\}.$$

In bit operations, the last set is obtained by shifting D_{c-k} right by $b - v$. The shift sends the bit for $x - v$, namely $b + x - v$, to bit x. If any future mask becomes full, meaning that all n row values are forbidden for some future column, the branch is pruned.

2.2 Symmetry and Prefix Decomposition

The implementation uses two levels of D_4 symmetry handling. During prefix generation it applies inexpensive first-entry constraints. At completion it applies the full D_4 lexicographic test and counts the size of the orbit.

The eight transforms of a completed permutation p are represented as

$$\begin{aligned} T_0(i) &= p_i, & T_1(i) &= n-1-p_i, \\ T_2(i) &= p_{n-1-i}, & T_3(i) &= n-1-p_{n-1-i}, \\ T_4(i) &= p^{-1}(i), & T_5(i) &= n-1-p^{-1}(i), \\ T_6(i) &= p^{-1}(n-1-i), & T_7(i) &= n-1-p^{-1}(n-1-i). \end{aligned}$$

A completed array is accepted only if T_0 is lexicographically minimal among these eight sequences. If it is accepted, the solver adds the number of distinct transforms among $T_0, \ldots, T_7$ to the count.

The first-entry constraints are the consequences of this lexicographic rule that can be enforced immediately after $p_0 = a$ is chosen. They are:

$$0 \le a \le \left\lfloor \frac{n-1}{2} \right\rfloor,$$

$$a \le p_{n-1} \le n-1-a,$$

and the values 0 and $n-1$ may occur only in columns

$$a,\ a+1, \ldots,\ n-1-a.$$

For $n = 28$, this means the first value a ranges from 0 to 13. In mask form, after choosing $p_0 = a$, the solver forbids rows 0 and $n-1$ in columns $1, \ldots, a-1$ and $n-a, \ldots, n-1$, and restricts the final column to rows $a, \ldots, n-1-a$.

The prefix phase then runs the same placement update as above until depth $m = 6$. Only the six row values

$$p_0, \ldots, p_5$$

are stored for each prefix. The full masks D_h and F_c are rebuilt when the prefix is later solved. For order 28, this prefix generation phase produces

$$73{,}949{,}504$$

valid depth-6 states under the first-entry D_4 constraints and forward-checking nonemptiness tests. These states are independent subproblems and therefore provide the unit of sampling and distributed execution.

To rebuild a stored prefix $p_0, \ldots, p_{m-1}$, the solver initializes all masks to zero, reapplies the first-entry constraints from p_0, inserts every prefix difference

$$p_x - p_y, \qquad 0 \le y < x < m,$$

into D_{x-y}, and recomputes, for every $c \ge m$,

$$F_c = S_c \cup \{p_0, \ldots, p_{m-1}\} \cup \{x : x - p_i \in D_{c-i} \text{ for some } 0 \le i < m\}.$$

The rebuilt state is then exactly the state that would have been obtained by continuing the original depth-first search to depth m.

2.3 Specialized Tail Solver

After a prefix is rebuilt, the remaining search is the following recursive procedure. Here $A = (1 \ll n) - 1$ is the mask of all row values.

SEARCH(k, F, D, p) :
  if $k = n$, add the D_4 orbit size of p if p is canonical, and return;
  $V \leftarrow \neg F_k \;\&\; A$;
  for each set bit v of V :
    reject v if $v - p_{k-h} \in D_h$ for some $1 \le h \le k$;
    otherwise insert these k new differences into D;
    form $F'_c = F_c \cup \{v\} \cup \{x : x - v \in D_{c-k}\}$ for $c > k$;
    reject the branch if some F'_c is full;
    apply the late two-row support test when enabled;
    recurse to SEARCH$(k + 1, F', D, p)$.

The implementation generates a separate function for each level:

$$\texttt{solve\_s\_6}, \texttt{solve\_s\_7}, \ldots, \texttt{solve\_s\_28}.$$

For $6 \le k < 28$, function `solve_s_k` places column k, saves only the modified masks $D_1, \ldots, D_k$, updates the state by straight-line code, and calls the next function. The terminal function `solve_s_28` performs the canonical-orbit test. Specialization removes generic recursion dispatch and many array-indexed inner-loop operations.

2.4 Limited Two-Row Support Pruning

The additional pruning rule in the primary implementation is a limited two-row support test. It is applied after placing column k for

$$18 \le k < 28,$$

to the remaining columns $s = k + 1, \ldots, n - 1$. Let

$$C_a = \{x : x \notin F_a\}$$

be the current candidate set for an unfilled column a. For two unfilled columns $a \ne b$, define compatibility by the immediate Costas condition against the already used differences:

$$\mathrm{comp}(a, x, b, y) \iff x \ne y \text{ and } \begin{cases} y - x \notin D_{b-a}, & a < b, \\ x - y \notin D_{a-b}, & b < a. \end{cases}$$

Thus a value $x \in C_a$ has support in column b if there exists $y \in C_b$ with $\mathrm{comp}(a, x, b, y)$.

The check is:

```
TwoRow(s, F, D) :
  repeat
    changed ← false;
    for a = s, ..., n − 1 :
      if C_a = ∅, return false;
      if |C_a| > 4, continue;
      R ← ∅;
      for x ∈ C_a :
        if some b ≠ a gives no support for x, add x to R;
      F_a ← F_a ∪ R;
      if R ≠ ∅, set changed ← true;
  until changed is false;
  for every pair a < b of remaining columns:
    if no compatible pair (x, y) ∈ C_a × C_b exists, return false;
  return true.
```

Only columns with $|C_a| \leq 4$ are filtered in the fixed-point deletion phase. This is the "limited" part of the rule. The final pair-existence scan is still applied to every pair of remaining columns.

Proposition 1 (Soundness of the limited two-row test)**.** *If the two-row procedure returns false, then the current partial permutation has no completion. Removing a value x during the fixed-point phase also preserves all possible completions.*

Proof. Suppose a completion uses x at column a. For every other unfilled column b, the completion assigns some value $y \in C_b$, with $y \neq x$, and the new pair of dots at columns a and b cannot repeat an already used difference of horizontal length $|a - b|$. Therefore x must have support in every other column. If some column gives no support, no completion can use x, so deleting x is sound. After all such deletions, if two columns a, b have no compatible pair at all, then no completion can assign values to both columns. Hence returning false is sound. □

Other rules were implemented and then rejected on cost rather than on correctness. Singleton propagation, small Hall-type checks, matching-based capacity tests, and late dynamic row choice are valid search reductions or ordering changes, but were slower in the tested implementations: a rule must save more downstream work than it costs in bitset operations, support scans, and bookkeeping.

2.5 Fixed-Order Incremental-D4 Variant

For comparison we also timed an independent CPU implementation of the same fixed-column Costas backtracking search. This variant enforces D_4 canonicality incrementally. For each nontrivial dihedral transform, it compares entries from position zero and stops if the next transformed entry is not yet determined. A branch is discarded only when all earlier compared entries are determined and equal and the first unequal entry makes the current partial permutation lexicographically larger. Such a branch cannot become the minimal representative of its orbit.

The variant also uses forward-checking masks for future columns, vectorized over several columns at a time, and a scalar specialized tail for the last few positions. In the benchmarked build the order was fixed at compile time to $n = 28$, allowing loop bounds and masks to be optimized for that order. This implementation provides the second CPU timing line in Appendix A.

2.6 Validation and Timing Protocol

Correctness was checked by running all solver variants at orders $8 \le n \le 20$, where the results agree with published enumeration records and database tables [8, 11, 3]. For order 28, runtime was estimated from a fixed-seed random sample of depth-6 subtrees. The two-row-cut implementation gives an estimate of 54.03 single-core CPU years; Appendix A reports the complete benchmark protocol and the second fixed-$n = 28$ CPU variant.

3 GPU Split Search for Order 30

The order-30 CUDA solver uses the CPU transition rule with a different work decomposition. The host enumerates reduced depth-6 prefixes, the GPU expands them to depth 10, and one warp searches each subtree through column 27. The two remaining assignments are then checked exactly, followed by the usual D_4 orbit test. Related hardware implementations include FPGA searches for Costas arrays and GPU/FPGA searches for multidimensional variants [7, 1].

3.1 Work Decomposition

The split parameters are

$$n = 30, \qquad m = 6, \qquad t = 10, \qquad s = 28,$$

where m is the host prefix depth, t is the emitted GPU work-unit depth, and s is the frontier depth at which survivors receive the exact two-column tail check. The measured number of reduced depth-6 prefixes is

$$P_6(30) = 120{,}969{,}637.$$

Expansion from depth 6 to depth 10 produces

$$M_{10}(30) = 6{,}676{,}844{,}782{,}690$$

valid depth-10 prefixes before the lightweight prefix-canonical filter described below. This is the generated frontier count used in the conservative timing projection. It already includes the same first-entry mirror reduction used in the CPU search: the first value is restricted to the lower half of the row range. Therefore no additional factor of two is applied in the projections; the final D_4 orbit rule accounts for the symmetry at completion.

For a fixed depth-6 prefix, the raw four-column extension space has size

$$24 \cdot 23 \cdot 22 \cdot 21 = 255{,}024.$$

The Costas constraints and forward-checking masks reduce this to a measured mean of

$$\frac{M_{10}(30)}{P_6(30)} = 55{,}194.39$$

valid depth-10 children per depth-6 prefix. The reduced depth-6 list is shuffled before ranges are selected, so the timed ranges give a more representative sample than contiguous lexicographic ranges.

3.2 Search State

The GPU solver uses the state and placement update of Subsection 2.1. For $n = 30$, F_c is a 30-bit row-value mask and D_h is a shifted 64-bit mask; the difference $p_k - p_{k-h}$ is stored at offset

$$31 + p_k - p_{k-h}.$$

The depth-6 generator, expansion kernel, depth-10 rebuild, and warp-owned search all use this transition.

3.3 Expansion to Depth 10

For a batch of depth-6 records, the expansion kernel maps

$$\{\text{depth-6 prefix}\} \times 24 \cdot 23 \cdot 22 \cdot 21$$

to candidate ten-column records. Each GPU thread decodes one four-position code into an ordered choice of unused row values for columns $6, 7, 8, 9$, rebuilds D and F from the six-column prefix, applies the transition above, and returns if a candidate value is forbidden or if any future column becomes impossible.

After column 9 is placed, a surviving prefix is also passed through a lightweight D_4 prefix-canonical filter. For each nonidentity transform T_j, the filter compares the known sequence

$$p_0, \ldots, p_9$$

with the beginning of T_j from position zero. The comparison continues only while the next transformed entry is determined by the prefix; at the first undetermined entry, the comparison stops and the prefix is kept. For example, inverse-type transforms can be evaluated at position i only if row value i, or $n-1-i$, has already appeared in the prefix; reversal-type transforms are unknown at the beginning because they depend on final columns. The prefix is discarded only when all earlier compared entries are determined and equal and the first unequal entry makes the prefix larger than T_j. Every completion of such a prefix is lexicographically larger than its T_j-image and therefore cannot be the D_4-canonical representative. If the prefix survives, the expansion kernel appends only

$$(p_0, \ldots, p_9)$$

to the depth-10 output stream. No D or F state is stored in global memory with the work unit; the solve kernel rebuilds those masks deterministically from the ten row values.

The expansion kernel runs before the warp search. Appendix A records the measured split between expansion and solve time.

3.4 Warp-Owned Depth-28 Search

The solve kernel assigns one CUDA warp to one depth-10 subtree:

$$\text{one CUDA warp} \quad \longleftrightarrow \quad \text{one depth-10 subtree.}$$

The mutable search state p, D, F, rollback masks, valid-value stack, and unused-value stack are stored in shared memory for each warp. Lane 0 controls candidate selection and stack movement. The other lanes cooperate on difference-mask updates, future-forbidden updates, and empty-domain checks. Warps pull depth-10 jobs from a persistent global queue until the batch is exhausted.

This layout was motivated by an earlier scalar GPU kernel in which one thread owned one depth-10 subtree. That direct mapping kept the depth-first search state in thread-local storage, and the compiler could not keep the state in registers without spilling. In the RTX 5090 build, the scalar kernel for the depth-10 to depth-28 search used 255 registers and spilled to local memory. By moving the mutable state to per-warp shared memory, the warp-owned kernel used 48 registers, a zero-byte stack frame, no spills, and 24,336 bytes of shared memory per block.

The depth-first search uses the fixed column order

$$10, 11, \ldots, 27.$$

Rollback is incremental: the solver stores the newly added future-forbidden and difference bits for a placement and clears exactly those bits when the branch returns. If a valid assignment

through column 27 is found, only two values remain unused. The solver checks the two possible assignments to columns 28 and 29 against the final future-forbidden masks and then applies the full D_4 canonical-orbit test.

Each emitted depth-10 prefix belongs to one depth-6 prefix, and rebuilding its masks gives the same state as sequential search. The prefix-canonical filter rejects only branches already known to be noncanonical; the final orbit test is unchanged. The split therefore preserves the sequential search tree and its orbit-counted total.

The benchmark in Appendix A times the depth-10 to depth-28 traversal for prefixes retained by the canonical filter.

4 Local Difference-Triangle Signature Transport

The method uses local patterns from known Costas arrays to restrict searches at larger orders. From a block of consecutive first differences, we record the absolute vertical displacements over every subinterval, grouped by horizontal distance. A target prefix is kept only if its signature occurs in the chosen source data. Related structural work also uses the difference-triangle representation of Costas arrays [13, 5].

4.1 Difference-Triangle Coordinates

Let $\mathcal{C}_n$ be the set of Costas permutations of order n, written as permutations of $\{0, \dots, n-1\}$. For $p \in \mathcal{C}_n$, define its first-difference word

$$\Delta p = (a_0, \dots, a_{n-2}), \qquad a_i = p_{i+1} - p_i.$$

The entries of the usual Costas difference triangle are consecutive sums of this word:

$$T_\ell(i) = \sum_{t=i}^{i+\ell-1} a_t = p_{i+\ell} - p_i, \qquad 1 \le \ell \le n-1, \quad 0 \le i < n-\ell.$$

Thus row ℓ of the triangle records all vertical displacements with horizontal displacement ℓ.

Proposition 2 (Reconstruction from first differences)**.** *Let $a = (a_0, \dots, a_{n-2}) \in \mathbb{Z}^{n-1}$, and set*

$$s_0 = 0, \qquad s_j = \sum_{i=0}^{j-1} a_i \quad (1 \le j \le n-1).$$

Then $a = \Delta p$ for a permutation p of $\{0, \dots, n-1\}$ if and only if the n integers $s_0, \dots, s_{n-1}$ are distinct and

$$\max_j s_j - \min_j s_j = n - 1.$$

In that case the permutation is

$$p_j = s_j - \min_i s_i.$$

Proof. If $a = \Delta p$, then $s_j = p_j - p_0$. The values s_j are distinct because the values p_j are distinct, and their range is $n-1$ because $\{p_j\} = \{0, \dots, n-1\}$. Conversely, if the prefix sums are distinct and have range $n-1$, then translating them by $-\min_i s_i$ gives exactly the set $\{0, \dots, n-1\}$. The displayed formula therefore defines a permutation, and its first differences are the entries of a. □

The Costas condition is equivalent to pairwise distinct entries in every row T_ℓ: equal entries in one row give equal vectors with horizontal displacement ℓ, while vectors from different rows have different horizontal displacements.

4.2 Local Absolute-Set Signatures

For a finite integer word $w = (w_0, \ldots, w_{d-1})$, define

$$A_\ell(w) = \left\{ \left| \sum_{t=i}^{i+\ell-1} w_t \right| : 0 \le i \le d-\ell \right\}, \qquad 1 \le \ell \le d.$$

The depth-d absolute-set signature of w is

$$\alpha_d(w) = \big(A_1(w), A_2(w), \ldots, A_d(w)\big).$$

This signature forgets the order of entries inside each triangle row and also forgets signs. It retains the local collection of displacement magnitudes created by the word.

For every word w of length d,

$$\alpha_d(w) = \alpha_d(-w) = \alpha_d(w^{\mathrm{rev}}),$$

where $w^{\mathrm{rev}} = (w_{d-1}, \ldots, w_0)$. Negation changes only the signs of the sums, and reversal changes only their order within each set $A_\ell(w)$.

For each order n, define the prefix-signature set

$$L_n(d) = \{\alpha_d(a_0, \ldots, a_{d-1}) : (a_0, \ldots, a_{n-2}) = \Delta p,\ p \in \mathcal{C}_n\}.$$

The experiments below use this prefix-signature set because the filter can then be applied early in the target search. The same definition can be made for all consecutive windows of length d, but that is a different experiment.

4.3 Cross-Order Filtering

For source order S, target order T, and signature depth d, define

$$h_d(S,T) = |\{q \in \mathcal{C}_T : \alpha_d(b_0, \ldots, b_{d-1}) \in L_S(d),\ \Delta q = (b_0, \ldots, b_{T-2})\}|.$$

This counts target arrays whose first d differences have a signature seen at source order S. When both sets are complete,

$$h_d(S,T) > 0 \quad \Longleftrightarrow \quad L_S(d) \cap L_T(d) \neq \emptyset.$$

Each run reads signatures from a source file, tests them in the solver's search orientation, and outputs the full D_4 orbit of each accepted array. Let $E_d(S,T)$ be the set of distinct rows returned by this run and

$$e_d(S,T) = |E_d(S,T)|.$$

The experiments report e_d, which need not equal h_d unless the source and orientation conventions match the complete sets above. A value of $e_d(S,T) = 0$ means that this source file selected no target array; it does not imply that order T has no Costas arrays. Set intersections below refer to the returned sets $E_d(S,T)$.

4.4 Depth-6 Results

The measurements in this subsection use the absolute-set signature with $d = 6$. Since a depth-6 signature uses six first differences, the source order must be at least 7. We tested all pairs

$$7 \le S < T \le 28.$$

All searches used the same settings: prefix decomposition depth 5, no lookahead, and full orbit output under D_4.

Source S	Target T	Valid outputs	Known target count
13	14	628	17,252
15	16	256	21,104
17	18	124	15,096

Table 1: Calibration runs for the depth-6 absolute-set signature.

Table 1 shows why this signature depth was used. It removes most target arrays in small-order tests while still producing valid arrays. The target counts in the table are standard enumeration counts [8, 11, 3].

We ran all 231 pairs $7 \le S < T \le 28$. The verification pass found no missing or extra runs and checked the process return code and output-line count for every pair. For successful runs, it also checked every output independently for the Costas property and confirmed the recorded totals. Of the 231 order pairs, 63 produced arrays, with 5,512 valid outputs in total. No search using source order 7 returned an array.

Search result	Value
Tested order pairs $7 \le S < T \le 28$	231
Completed runs	231
Missing or extra runs	0
Independent output-check failures	0
Successful order pairs	63
Total valid outputs	5512
Targets with no output	$24, 25, 26, 27$
Sources producing order-28 outputs	$26, 27$

Table 2: Verified depth-6 cross-order search results through order 28.

Changing the source order can change the selected part of the target space, not only the number of selected arrays. For target order 16, the depth-6 filtered counts were

$$e_6(13, 16) = 136, \qquad e_6(14, 16) = 192, \qquad e_6(15, 16) = 256.$$

The corresponding selected sets had intersections

$$|E_6(13, 16) \cap E_6(14, 16)| = 16, \qquad |E_6(13, 16) \cap E_6(15, 16)| = 0,$$

and

$$|E_6(14, 16) \cap E_6(15, 16)| = 16.$$

Their union had size 552, and 120 arrays selected by source order 13 were selected by neither source order 14 nor source order 15. Thus the prefix-signature sets are not simply nested by order.

Target order 22 shows the same pattern. Among source orders $7, \ldots, 21$, the only successful depth-6 searches were

$$e_6(13, 22) = 8, \qquad e_6(21, 22) = 16.$$

All sources

$$7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20$$

returned no arrays for this target.

No source order produced an array at targets $24, 25, 26, 27$. For target order 28, sources 26 and 27 produced 8 and 16 outputs, respectively:

$$e_6(26, 28) = 8, \qquad e_6(27, 28) = 16.$$

Source S	Target T	Valid outputs
21	22	16
21	23	8
22	23	20
26	28	8
27	28	16

Table 3: Successful high-order searches through order 28.

Table 3 records the successful high-order searches.

We also compared the outputs with generated-family labels in the Beard database [3]. After grouping the 5,512 outputs into 700 target orbits, 663 have no generated-family label in the database and 37 are classified. Among the classified orbits, successful searches include Welch sources leading to Welch targets, Taylor sources to Taylor targets, and Rickard or Lempel sources to Taylor targets. The searches $21 \to 22$ and $27 \to 28$ both have Welch source and target labels; at $22 \to 23$, Taylor and Rickard source labels lead to Taylor target labels.

The results are sparse and not monotone in $|T - S|$: a nearby source order is not necessarily a better filter than a distant one.

5 Conclusions

The CPU solver agrees with known counts for $8 \le n \le 20$ and gives an order-28 estimate of 54.03 single-core CPU years. The order-30 GPU traversal projects to 7.97 RTX 5090 years, or about one year on eight identical GPUs under ideal splitting.

The local-signature experiment showed that patterns extracted from smaller Costas arrays can guide searches at larger orders. Across the 231 tested order pairs through order 28, the restricted searches returned 5,512 valid Costas arrays for 63 order pairs. In particular, patterns from orders 26 and 27 produced valid Costas arrays of order 28. Success does not vary regularly with the difference in order, so nearby orders are not necessarily the most useful sources.

A Benchmark Details for CPU and GPU Estimates

This appendix records the timing measurements used for the CPU and GPU projections, together with the estimation formulas and benchmark conditions.

A.1 CPU Order-28 Prefix Timing

Both CPU estimates use depth-6 prefix sampling at order 28. For a prefix population of size P, a sample of size r, and elapsed single-prefix times t_i, the reported estimate is

$$\widehat{Y} = \frac{P}{r} \frac{\sum_{i=1}^{r} t_i}{365.25 \cdot 24 \cdot 3600},$$

in single-core CPU years. The per-prefix times are wall seconds for independent single-threaded subtrees; concurrent workers only reduce calendar time for the sampling run.

We timed each solver on an independent random sample of 1,000 valid depth-6 prefixes from its order-28 work partition. This gives a direct estimate of the total single-core time required by that solver.

The CPU system was an AMD Ryzen 9 9950X3D with 16 cores and 32 hardware threads, running Linux 7.0.9-arch2-1. The compiler was GCC 16.1.1, dated 2026-04-30. Both programs were built with `-O3`, `-march=native`, link-time optimization, and OpenMP support; the second build additionally used `-DN_FIXED=28`. Each experiment used 1,000 sampled depth-6 prefixes.

Implementation	P	r	Mean s/prefix	CPU years	Bootstrap 95% CI
two-row cut	73,949,504	1,000	23.055275	54.025849	[52.159254, 55.871053]
fixed-n D4 variant	73,444,803	1,000	41.978757	97.698226	[94.490709, 100.932794]

Table 4: Order-28 CPU single-core time estimates from 1,000 depth-6 samples.

Implementation	SD s	CV	p50 s	p90 s	p99 s	max s
two-row cut	12.880290	0.558670	22.490646	39.605893	54.022900	89.692249
fixed-n D4 variant	22.481668	0.535549	40.222218	73.471360	101.037794	119.018943

Table 5: Distribution summaries for the sampled CPU prefix times.

The sampled first values p_0 are shown in Table 6. The row values are zero-based, matching the rest of the paper.

First value p_0	two-row cut	fixed-n D4	First value p_0	two-row cut	fixed-n D4
0	94	69	7	61	63
1	104	102	8	40	62
2	98	87	9	65	58
3	84	78	10	80	65
4	87	85	11	53	70
5	62	71	12	56	54
6	66	65	13	50	71

Table 6: First-row distribution in the 1,000 sampled CPU prefixes.

A.2 GPU Order-30 Throughput Calibration

The GPU benchmark was run on an NVIDIA GeForce RTX 5090 with 32,607 MiB of memory, driver 580.126.09, CUDA 12.8, and a 600 W power limit. The benchmark used the warp-owned CUDA implementation described above. The launch parameters were

$$\texttt{start\_d6} = 0, \quad \texttt{count\_d6} = 5040, \quad \texttt{batch\_d6} = 24,$$
$$\texttt{expand\_block} = 128, \quad \texttt{solve\_block} = 256, \quad \texttt{seed} = 20260531,$$
$$\texttt{warp\_mult} = 4.$$

The program enumerates valid reduced depth-6 prefixes, shuffles them with the seed, and takes the requested range from that shuffled list. Thus the 5,040 prefixes form a random sample without replacement from the enumerated depth-6 population.

Here "kept" means that the depth-10 prefix survived the prefix-canonical filter and was sent to the depth-28 solve kernel. "Generated" means kept plus prefix-filter rejected. The aggregate solve rate is therefore a kept-job solve rate. The projection below divides the full generated frontier $M_{10}(30)$ by this kept-job rate:

$$\frac{6{,}676{,}844{,}782{,}690}{26{,}538.917 \cdot 365.25 \cdot 24 \cdot 3600} = 7.972.$$

This is conservative for the measured split, because prefix-filter rejected jobs do not enter the solve kernel and the measured expansion time is negligible relative to solve time. Per-batch rates over the 210 batches had mean 26,744.815 depth-10 jobs/s, standard deviation 1,821.101, minimum 22,667.740, and maximum 32,474.422. The larger 5,040-prefix run is consistent with the earlier 1,920-prefix calibration, which projected 7.974 single-RTX-5090 GPU years.

Quantity	Value
Depth-6 prefixes	5,040
Batches	210
Kept depth-10 jobs	266,496,139
Prefix-filter rejected depth-10 jobs	13,522,830
Generated depth-10 jobs	280,018,969
Expansion time	0.403428 s
Solve time	10,041.711053 s
Mean kept jobs per depth-6 prefix	52,876.22
Aggregate solve rate	26,538.917 depth-10 jobs/s
Generated frontier $M_{10}(30)$	6,676,844,782,690
Projected single RTX 5090 time	7.972 GPU years
Projected eight RTX 5090 time	0.997 calendar years

Table 7: RTX 5090 5,040-prefix throughput benchmark for the order-30 GPU split.

References

[1] Rafael A. Arce-Nazario and José R. Ortiz-Ubarri. Multidimensional costas arrays and their enumeration using gpus and fpgas. *International Journal of Reconfigurable Computing*, 2012:1–9, 2012.

[2] Lionel Barker, Konstantinos Drakakis, and Scott T. Rickard. On the complexity of the verification of the costas property. *Proceedings of the IEEE*, 97(3):586–593, 2009.

[3] James K. Beard. Costas arrays and enumeration to order 1030, 2017. Database.

[4] James K. Beard, Jon C. Russo, Keith G. Erickson, Michael C. Monteleone, and Michael T. Wright. Costas array generation and search methodology. *IEEE Transactions on Aerospace and Electronic Systems*, 43(2):522–538, 2007.

[5] Bill Correll and Christopher N. Swanson. Difference-based structural properties of costas arrays. *Designs, Codes and Cryptography*, 91:779–794, 2023.

[6] John P. Costas. A study of a class of detection waveforms having nearly ideal range-doppler ambiguity properties. *Proceedings of the IEEE*, 72(8):996–1009, 1984.

[7] Jim Devlin and Scott T. Rickard. Accelerated costas array enumeration using fpgas. In *42nd Annual Conference on Information Sciences and Systems*, pages 1252–1257, 2008.

[8] Konstantinos Drakakis. A review of costas arrays. *Journal of Applied Mathematics*, 2006:1–32, 2006.

[9] Konstantinos Drakakis, Francesco Iorio, and Scott T. Rickard. The enumeration of costas arrays of order 28 and its consequences. *Advances in Mathematics of Communications*, 5(1):69–86, 2011.

[10] Konstantinos Drakakis, Francesco Iorio, Scott T. Rickard, and John Walsh. Results of the enumeration of costas arrays of order 29. *Advances in Mathematics of Communications*, 5(3):547–553, 2011.

[11] Solomon W. Golomb and Guang Gong. The status of costas arrays. *IEEE Transactions on Information Theory*, 53(11):4260–4265, 2007.

[12] Solomon W. Golomb and Herbert Taylor. Constructions and properties of costas arrays. *Proceedings of the IEEE*, 72(9):1143–1163, 1984.

[13] Jonathan Jedwab and Jane Wodlinger. Structural properties of costas arrays. *Advances in Mathematics of Communications*, 8(3):241–256, 2014.

[14] Jon C. Russo, Keith G. Erickson, and James K. Beard. Costas array search technique that maximizes backtrack and symmetry exploitation. In *44th Annual Conference on Information Sciences and Systems*, pages 1–6, 2010.